Implementation of MPEG-4 on the Philips Co Vector Processor

B. An[1,2] S. Balakrishnan[2] C.H. van Berkel[2] D. Cheresiz[1,*] B. Juurlink[1] S. Vassiliadis[1]

[1] *Computer Engineering Laboratory*
Delft University of Technology
2600 GA Delft
The Netherlands

[2] *Philips Research Laboratories*
Prof. Holstlaan, 4
5656 AA Eindhoven
The Netherlands

*email: cheresiz@dutepp0.et.tudelft.nl

Abstract— **Multimedia applications provide new highly valuable services to the consumer and form, consequently, a new important workload for desktop systems. The increased computing power of the embedded processors required in baseband processing for new high-bandwidth wireless communication protocols (e.g UMTS, CDMA-2000) can make multimedia processing possible also for the mobile devices, such as cell phones. These devices must meet high performance requirements of multimedia applications while maintaining low cost and low power consumption.**

As a new platform which can provide the necessary computing power for processing the inner tranciever functions of the modem for UMTS, Philips develops a novel processor, called Co Vector-Processor (CVP), which is based on the following techniques. First, CVP exploits the data-level parallelism (DLP) by processing 256-bit data items, which are interpreted as vectors consisting of 32 8-bit, 16 16-bit, or eight 32-bit elements. Therefore, a single vector instruction of CVP performs up to 32 operations. Second, CVP also exploits the instruction-level parallelism (ILP) using VLIW approach: several vector instructions are packed in a single very long instruction word (VLIW), and are executed in parallel.

This high-performance machine can be employed not only for its original purpose, the baseband processing, but also for multimedia processing. In this paper we investigate how well the parallel processing capabilities of CVP can be utilized for a typical media application, and estimate the performance levels which can be achieved. We use the MPEG-4 video encoder as a benchmark. We identify two most time-consuming kernels of this application, the Motion Estimation (ME) and the Discrete Cosine Transform (DCT), and rewrite them using CVP instructions. These kernels operate on 8×8 pixel blocks. We propose two different storage schemes: half-block based and pixel based. Additionally, we propose some architecture extensions, such as employing full-shuffles. We show that by using the appropriate storage schemes and the proposed extensions the performance of ME and DCT can be improved by factors of 2.88 and 1.84, respectively. We, therefore, show that the most important kernels of MPEG-4 encoder can be vectorized and efficiently implemented on CVP.

Keywords: processor architecture, video processing, vectorization

I. Introduction

Importance of wireless communication services is rapidly increasing and demand for new services with higher quality is growing. To make such services feasible, third generation (3G) wireless communication standards are developed, such as UMTS/TDD, UMT/FDD and TD-SCDMA. These standards provide higher communication bandwidth than 2G standards, for example, UMTS (a 3G standard) requires more than 10 times higher data bandwidth (384 kbps) than the GSM system (about 30kbps) and, consequently, impose higher requirements on the processing power of a mobile device.

An architecture which would be powerful and flexible enough to meet the 3G standards requirements with low cost and low power consumption is developed within the *SW-modem for 3G mobiles* project carried out at Philips Nat.Lab. The aim of this project is to develop a base-band platform for 3G mobile terminals which supports multiple 3G (and 2G) standards and modes, inter-standard hand-over, and a variety of media applications and functions. The proposed platform is based on a powerful Co Vector-Processor (CVP). The CVP combines vector (8×32, 16×16, or 32×8 bits) and scalar processing in a VLIW execution model to exploit data level parallelism and instruction level parallelism[1]. The functional units and the communication among these units can be configured. Powerful data processing can be provided by CVP and simple structure and low cost are achieved.

The CVP was originally designed for 3G-baseband processing, but since multimedia applications form another important workload for the 3G mobile terminals, suitability of CVP for media processing, e.g., audio and video processing, should be evaluated. The most probable standard for video processing for 3G mobile communications is MPEG-4.

The purpose of this paper is to investigate the feasibility of implementation on the CVP of the most compute-intensive kernels of the MPEG-4 encoder and decoder.

This paper is organized as follows. Section II presents a brief description of the CVP architecture. Section III describes one of the most compute-intensive kernels of MPEG-4, the motion estimation (ME), and two different storage organizations for MPEG-4 video frames: the *half-block* based (HB-based) and the *pixel scan order* based (PSO-based). In Section IV we describe the vectorization of the motion estimation for HB-based frames and study the performance of such implementations. In Section V the vectorization of the motion estimation for PSO-based frames is studied. In Section VI we investi-

gate how the DCT algorithm can be implemented on the CVP and study its performance. Finally, in Section VII some conclusions are drawn and directions for future work are proposed.

II. The CVP Architecture

The CVP can be applied in a system as presented in Figure 1. The CVP combines the advantages of VLIW and SIMD architectures to achieve high performance and operates under the supervision of a host DSP or micro-controller. The host processor can only assign tasks to the CVP, while the CVP can interrupt the host processor. The host processor and the CVP can run in parallel. To control the data traffic from and to the CVP, a DMA controller is employed. The standard bus interconnects the various blocks and the CVP is connected as a slave through it. All communication with the CVP is memory mapped via the vector memory (VM).

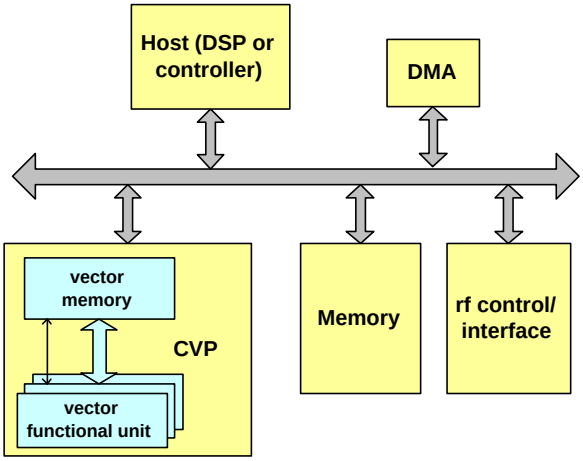

Fig. 1. A CVP-based 3G-modem hardware architecture

Vector parallelism, also known as *Single Instruction Multiple Data (SIMD)* parallelism, is exploited in the CVP. In this architecture This means that a single vector operation processes a vector of identical elements. The elements in CVP can be of the word- (8 bits), double-word (16 bits), or quad-word (32 bits) size. A word (8 bits) is the basic unit of memory addressing. The CVP typically stores and manipulates vectors of 32 words.

Orthogonally to vector parallelism, the instruction-level parallelism (ILP) is also exploited in CVP, in the form of the *Very Long Instruction Word (VLIW)* parallelism. Each CVP VLIW instruction consists of up to seven instructions which control seven corresponding functional units (FUs). These FUs are **IDU** (Instruction Distribution Unit), **VMU** (Vector Memory Unit), **CGU** (Code Generation Unit), **AMU**(Alu-Mac Unit), **SFU** (ShuFfle Unit), **SLU** (Shift Left Unit), and **SRU** (Shift Right Unit). These seven FUs operate in parallel. Each FU can receive and send both vector and scalar data. We can configure most FUs to tune their capabilities to the demands of specific algorithms.

The **IDU** consists of the program memory, and is responsible for sequencing the instructions, and distributing the seven instruction segments to itself and the six other FUs. It does not support branches, jumps, or interrupts. The **VMU** contains the *vector memory (VM)*. It can send a vector from the VM or receive a vector into the VM. There are also scalar send and receive operations. The VMU is the only FU connected to the external world, i.e. to the external bus. The **CGU** is the special-purpose unit used to generate vectors of CDMA code chips. In the studies reported in this paper it is not used. The **AMU** performs regular integer and fixed-point arithmetic. It supports inter-vector operations and intra-vector operations. In inter-vector operations, arithmetic is performed element-wise on multiple vectors, while in intra-vector operations, arithmetic is performed to the elements within a single vector[2]. The **SFU** allows to arbitrarily rearrange the elements of a vector according to a shuffle pattern. In the current implementation of CVP only half of the vector elements can be rearranged at a time, either even (even shuffle) or odd ones. Therefore, two shuffles, even and odd are executed, to perform one full shuffle operation. The **SLU** can shift the elements of the vector by a word, a double word or a quad word to the left. The data shifted out can be placed in SLU's scalar section. The shifted positions on the right side of the vector are padded either with zeroes or with the data taken from SLU's scalar section, depending on the type of SLU vector-operation issued[2]. The SRU is similar to the SLU, but shifts to the right.

Normally three to four FUs operate in parallel under control of a single VLIW instruction. Instructions are typically issued every clock cycle. Each FU has three sections: control, scalar (32 bits), and vector (256 bits). There is tight interaction between these sections (intra-FU communication) within an FU. The communication is strictly among the scalar sections and among vector sections (inter-FU communication) among FUs, as shown in Figure 2. The vector sections of all FUs except the IDU are connected by a full interconnect network. This feature allows each vector section to receive a vector from any of the other vector

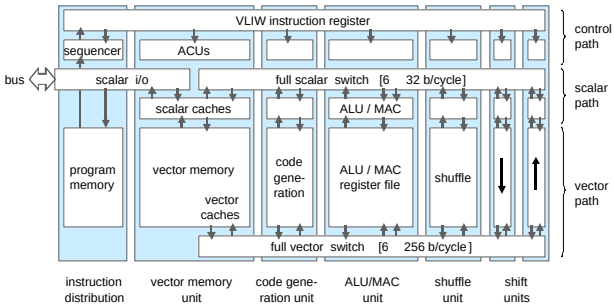

Fig. 2. CVP architecture

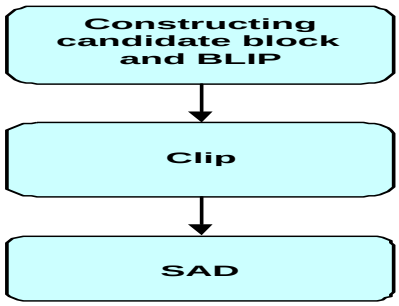

Fig. 3. Key functional blocks of the motion estimation

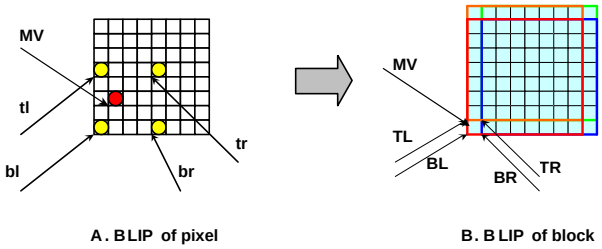

Fig. 4. Bilinear interpolation

sections during each cycle and enables the creation of arbitrary pipelines of the six FUs[2].

Many algorithms consist of parts that can be vectorized and parts that are of a scalar and sequential nature. It is not practical to involve a host processor since the interaction between these parts is often fine grained (every few clock cycles). The CVP allows these interactions to occur inside the FUs. There is considerable parallelism in the scalar path, in parallel to that of the vector path, as shown in Figure 2. The communication switch that connects the scalar sections of the FUs can be configured in a way that is identical to that of the vector sections[1]. The scalar sections are independent of vector sections.

For the current CVP architecture prototype, tools exist for assembly, simulation, and debugging. The simulator is bit-true and cycle true. In order to reduce the complexity of the programming task, a higher-level programming language, called CVP-C, which is a subset of C language, is developed to avoid error-prone tasks such as scheduling of FU operations and allocation of registers. The CVP-C programmer has to transform the original algorithm that operates on scalar samples into an algorithm operating on fixed-size vectors samples. The CVP-C compiler can then convert it to the CVP assembly and compound (or schedule) the operations which can be executed in parallel into VLIW CVP instructions.

III. Vectorization of Motion Estimation

Motion estimation, as well as many other important kernels of MPEG-4 operates on 8×8 pixel blocks. For a given *current block*, several *motion vectors*(MVs) are computed. A motion vector points to a lower-left corner of the corresponding *candidate block* and represent its displacement with respect to the current block. We remark that the candidate MVs are provided in such a way that the corresponding block fits into the *search region*, which is a domain of certain predefined shape around the current block. Then, for each MV the same sequence of operations, which is depicted graphically in Figure 3, is carried out.

First, the *candidate block* is constructed by means of *Bilinear Interpolation* (*BLIP*), as explained in the following. Let MV be a motion vector. The coordinates can be of the quarter-pixel accuracy, i.e. $\mathrm{MV} = (x + p, y + q)$, where $p, q \in \{0, 0.25, 0.5, 0.75\}$ and x, y are integer. In a case when $p = q = 0$, no interpolation is needed, and in case when $p = 0 \bigvee q = 0$, the candidate block is an interpolation of two blocks. In a general case, when $p \neq 0 \bigwedge q \neq 0$, the motion vector points inside an empty area surrounded by four pixels: tl, bl, tr and br, as shown in Figure 4(A). Four corresponding weighting factors (Atl, Abl, Atr, Abr) are assigned to these pixels. The bilinear interpolation of the pixel value that the MV points to is defined as the weighted sum of these four neighbor pixels:

$$\mathrm{BLIP} = \frac{tl \cdot Atl + bl \cdot Abl + tr \cdot Atr + br \cdot Abr}{Atl + Abl + Atr + Abr} \qquad (1)$$

The candidate block is constructed from the four blocks TL (top-left), BL (bottom-left), TR (top-right), and BR (bottom-right), which are located as depicted in Figure 4(B), by applying (1) to every four corresponding pixels of them.

After the candidate block is constructed, the *Clip* function restricts the pixel values resulting from BLIP to the $[10, 240]$ range, and, finally, the *Sum of Absolute Differences* (*SAD*) between the corresponding pixels

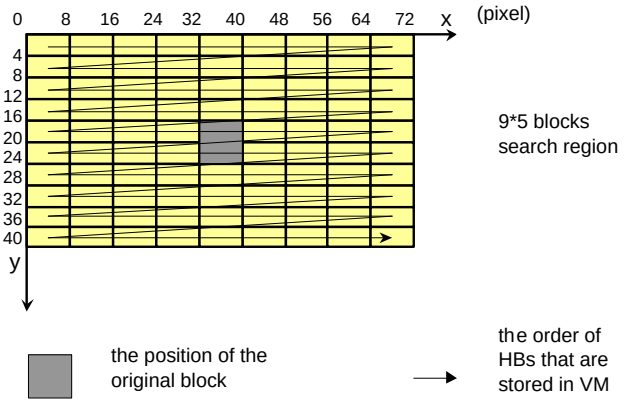

Fig. 5. HB-Based storage

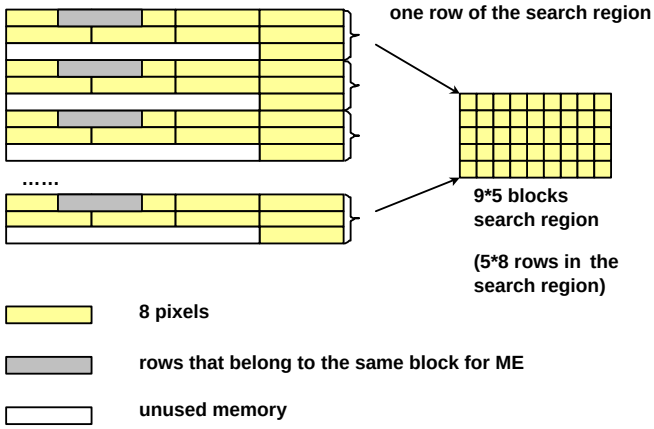

Fig. 6. PSO-Based storage

of the current and candidate block is computed. After the SADs for all candidate blocks have been computed, the motion vector corresponding to the block that minimizes the SAD is selected and passed to the later stages of the MPEG-4 encoding.

In this paper we assume that the *3DRS* algorithm for Motion Estimation (ME) is employed, where 7 candidate MVs are used to compute the ME for the current block. The search region is a rectangle centered at the current block and consisting of 5×9 blocks, or 40×72 pixels. We assume also that a CIF format video frame (352×288 pixels) is used.

Block Storage Schemes

The implementation of motion estimation greatly depends on the way in which frame is stored in the CVP's vector memory (VM). In the remainder of this section we propose two possible frame storage schemes: the *Half-Block Based* (*HB-based*) and the *Pixel Scan Order Based* (*PSO-based*).

Half-Block Based Scheme. We recall that a vector in CVP contains 32 bytes. Since a block contains 64 bytes, it can be stored in two vectors: the upper 4 rows in one vector, and the lower 4 rows in the other. The whole search region can be stored in 90 vectors. The half-blocks are stored in the memory in the row-major order, as shown in Figure 5.

Pixel-Scan-Order Based Scheme. As we described above, one row in the search region consists of $9 \times 8 = 72$ pixels. Hence, it can be kept in 3 consecutive vectors (96 words, i.e. 96 bytes) in the VM, with the last 24 words in the third vector being unused. All the 40 rows of the search region are stored, therefore, in 120 consecutive vectors in the VM, as depicted in Figure 6. Implementations of the motion estimation based on the proposed storage organizations are discussed in Section IV and Section V, respectively.

IV. ME Implementation for HB-Based frame

As we have explained above, to construct a candidate block, four blocks TL, BL, TR, and BR with integer coordinates should be fetched and interpolated. Below, we show the operations needed to fetch any of them. We observe that a block (with integer coordinates) is always contained in six half-blocks (HBs), as Figure 7 depicts. In Figure 8 we show how the upper HB of the desired block is extracted from HB1–HB4. From this figure, it can be easily observed that the pixels, which belong to the HB being constructed, can be located at arbitrary positions within the source HB/vector. To extract them, we employ the CVP shuffle operation $\circ$, defined as follows: if $a = b \circ c$, where a, b, and c are CVP vectors, then $a[i] = b[c[i]]$, if $0 < c[i] < 31$, and $a[i] = a[i]$ if $c[i] = -1$, which allows to extract arbitrary elements of b according to the *shuffle pattern* c, and insert them in the vector a. Each of HB1, HB2, HB3, and HB4 requires a distinct pattern for the necessary pixels to be extracted. So, four patterns and four full shuffle operations are needed to construct the upper half-block of the candi-

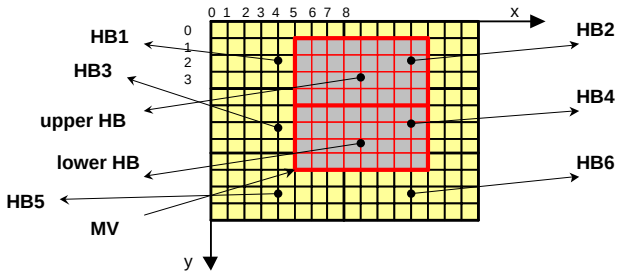

Fig. 7. One candidate block out of six HBs

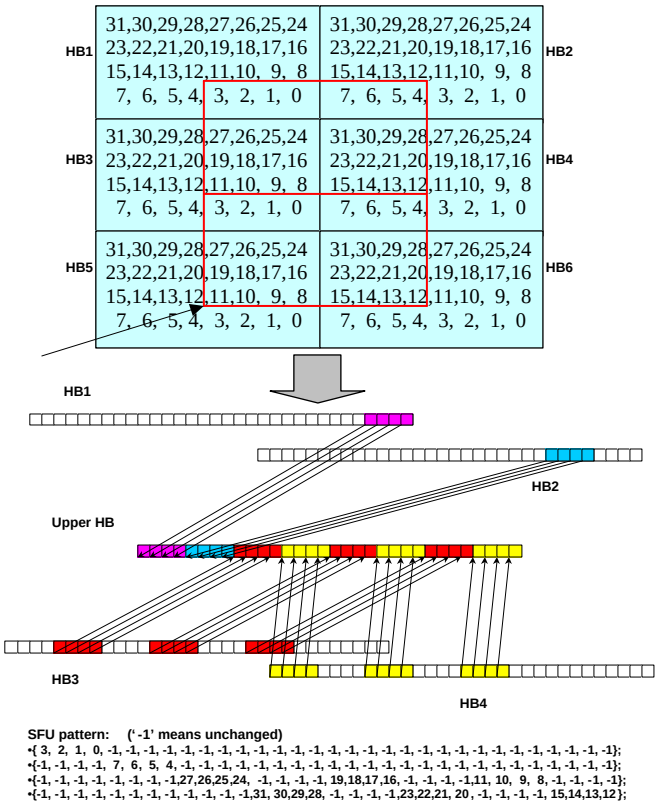

Fig. 8. Extraction of a block from six HBs

date block. We remark that the same patterns should be used construct the lower half-block from HB3-HB6. Therefore, 4 different SFU patterns and 8 full shuffle operations are required to construct the candidate block. Since a full vector shuffle is implemented by an odd shuffle and a consequent even shuffle on the current CVP, $8 \times 2 = 16$ SFU operations (*SFU ops*) are executed to form one block.

While four SFU patterns are needed to fetch, for example, the upper HB (UHB), the SFU unit has only two pattern registers (*sfu_cfg0*, *sfu_cfg1*). On the other hand, the same four patterns are needed to form the lower HB (LHB). Therefore, to avoid reloading of the patterns, the following approach is taken. Two patterns are loaded at one time. Instead of forming UHB completely, part of UHB and part of LHB are formed first. For example, HB1 and HB3 are shuffled to form UHB, and HB3 and HB5 for LHB using the two loaded patterns. After that, two other patterns are loaded to completely form UHB and LHB. However, after UHB is partially formed, the intermediate result has to be stored. After loading the other two patterns, the partially formed UHB should be used as the target (state) of the SFU operation. Unfortunately, a vector cannot be loaded into the implicit target register of the SFU (*sfu_state*) directly in the current CVP. Therefore, the partially formed UHB is copied (shuffled) from the SFU input register (*sfu_input*) to *sfu_state* using the copy pattern, thus requiring an extra SFU operation. Because the same copy operation is needed for the other two patterns, two extra SFU operations are needed to form UHB. Similarly, another two SFU ops are needed to form LHB. Totally, $16 + 4 = 20$ SFU ops are executed to form a block from six HBs.

We remark that since an MV can point anywhere inside an HB, which contains 32 pixels, there are 32 different cases when the MV is of pixel accuracy (i.e., it has integer coordinates). Hence, there are 32 different groups of SFU patterns and, as observed above, each group contains 4 patterns. These $4 \times 32 = 128$ patterns are stored in the VM as a lookup table. For different MVs, different group of patterns are loaded. From the MV we can determine which HBs in the VM contain the candidate block, which group of SFU patterns has to be used to form the block, and which group of weighting factors to be chosen for BLIP.

After forming the required blocks, BLIP is performed. To complete this, 8 MAC operations are executed, since 4 MAC operations are needed for BLIP for each of two half-blocks.

After BLIP, Clip function is performed to restrict pixel values to the range $[10, 240]$ instead of $[0, 255]$. Let v be the interpolated vector (half-block), $v_1 = (10, 10, \ldots, 10)$, and $v_2 = (240, 240, \ldots, 240)$. Then, by performing CVP operation $u = \text{MIN}(v, v_1)$ and, consequently $\text{MAX}(u, v_2)$, we will clip values of v to the desired range. Since interpolated block is stored in two vectors, four AMU operations are needed for clipping.

When pixels of the candidate block are clipped, the sum of the absolute differences (SAD) between this block and the original block is computed. SAD function is defined according to formula:

$$SAD(X, Y) = \sum_{i=0}^{7} \sum_{j=0}^{7} |X(i, j) - Y(i, j)| \qquad (2)$$

and is realized as follows. Let A, B be the upper HBs of the original and the candidate blocks, respectively. Two SUB and one MAX operation are performed to obtain the absolute differences of A and B and then intra add (IADD) is used to accumulate them and produce the SAD(A,B). Since blocks consist of two HBs, 9 AMU operations ((SUB×2 + MAX + DIADD)×2 + ADD) in total are used.

After obtaining the SAD values of the 7 candidate blocks and comparing these values, the block that has the smallest SAD value is selected as the matched block for ME. The corresponding MV of the matched block is stored for decoding and the ME of a block is completed.

Results

We now present the performance of the baseline ME implementation which was described above, and propose and evaluate several optimized ME implementations. The results are obtained as follows: first we implement the program in the CVP-C language, which is a subset of C developed specifically for the CVP. The compiler converts the code to VLIW CVP assembly instructions. Since each VLIW instruction can be executed in one cycle, the execution time of the VLIW code is equal to the number of VLIW instructions. This number is also referred to as the *length of the schedule*.

Table I summarizes the results for different ME implementations. The second column shows the number of *VMU-busy* cycles, i.e., cycles during which VMU was occupied. It also shows the VMU utilization, i.e., the ratio of the VMU-busy cycles to the total number of execution cycles. The third and the fourth columns depict number of busy cycles and utilization for the AMU and SFU units, respectively. The last column depicts the execution time and the speedup of each of ME implementations with respect to the baseline ME implementation described earlier in this section and referred to as ME1.

	VMU/UR	AMU/UR	SFU/UR	NoC/ME1*
ME1	55/35%	49/31%	**78/50%**	156/100%
ME2	45/37%	51/42%	**64/52%**	123/127%
ME3	31/27%	51/44%	**64/55%**	116/134%
ME4	**58/45%**	51/40%	40/31%	129/121%
ME5	31/29%	**51/48%**	32/30%	107/146%

TABLE I
PERFORMANCE OF ME IN HB-BASED FRAME

The baseline ME implementation, ME1, takes 156 cycles. The execution time is determined by the *critical path*, i.e., the longest chain of dependent operations. In our case, there are 80 SFU operations on the critical path. The shuffle unit is the most utilized (or *critical*) resource and constitutes the performance bottleneck.

Analysis of the ME1 implementation leads us to several SFU optimizations, which are depicted in Figure 9. The corresponding ME implementations are referred to as ME2–ME5.

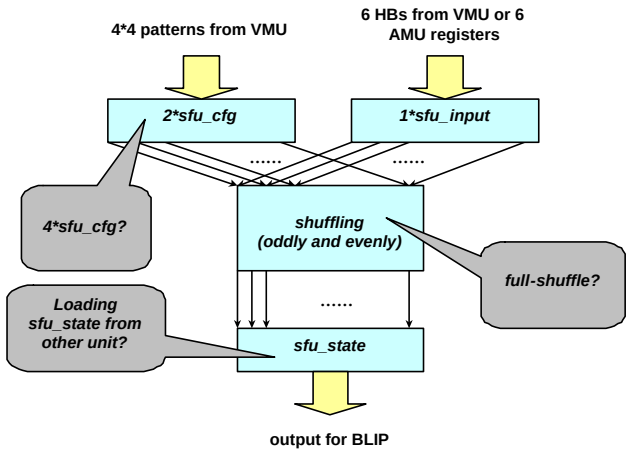

Fig. 9. SFU structure and suggestions for improvement

ME2: We recall that in ME1 a partially constructed HB (UHB or LHB) has to be loaded into *sfu_state*, the implicit target register in the SFU. Since there is no direct operation on this register, the partially formed HB from the SFU input register *sfu_input* is copied to *sfu_state* by executing an SFU op with the special pattern. If data can be loaded directly to the implicit SFU target register from the VM, the extra cycles are avoided. We refer to the approach with such a modification to the CVP as ME2. The ME2 implementation requires 123 cycles and attains a 27% speedup over ME1. The critical resource for ME2 is still the SFU with 64 SFU operations.

ME3: Another possibility to avoid the 16 SFU operations that copy the partially formed HB from *sfu_input* to *sfu_state* is to have four SFU configuration registers (*sfu_cfg*), instead of two. The CVP is extended with four *sfu_cfg* and the corresponding ME implementation is named ME3. It requires 116 cycles which corresponds to a speedup of 1.34 over ME1. The SFU is the critical resource with 64 SFU operations on the critical path.

ME4: Since the current CVP can only perform half-shuffle operation (odd shuffle and even shuffle), if full shuffle operation is available, just 40 SFU operations are needed to realize ME. The CVP is extended with the full shuffle operation and the corresponding ME implementation is called ME4. It takes 129 cycles, exhibiting a 21% speedup over ME1. We remark that the SFU is not the busiest FU anymore, and the VMU becomes the critical resource in.

ME5: This approach combines the ME3 and ME4 optimizations; i.e., the CVP is extended with full shuffle and four SFU configuration registers. It reduces the number of SFU operations from 80 to 32. However,

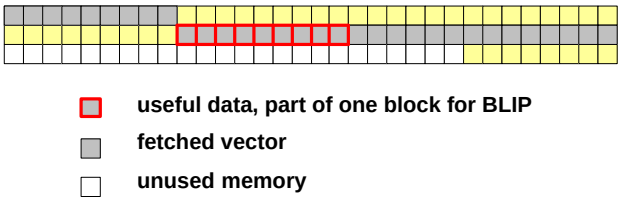

Fig. 10. Fetching vector using SLU

```
1. SLU:: RCV(slu0, VMU);
2. SLU:: SHIFT0(slu0, QWORD), RCV(slu0, SLU), SRCV(SLU);
3. SLU:: SHIFTS(slu1, QWORD), RCV(slu1, SLU);
4. SLU:: SHIFT0(slu0, QWORD), RCV(slu0, SLU), SRCV(SLU);
5. SLU:: SHIFTS(slu1, QWORD), RCV(slu1, SLU);
```

1. SHIFT0(slu0, QWORD) – shift slu0 left by 4 bytes, padd with 0s;
2. SHIFTS(slu1, QWORD) – shift slu1 left by 4 bytes, padd with scalar from the sslu
3. RCV(slu0, SLU) – receive vector from SLU and store it in slu0;
4. SRCV(SLU) – receive scalar from SLU and store it in sslu;

TABLE II
EXTRACTING TL/BL ROW USING SLU.

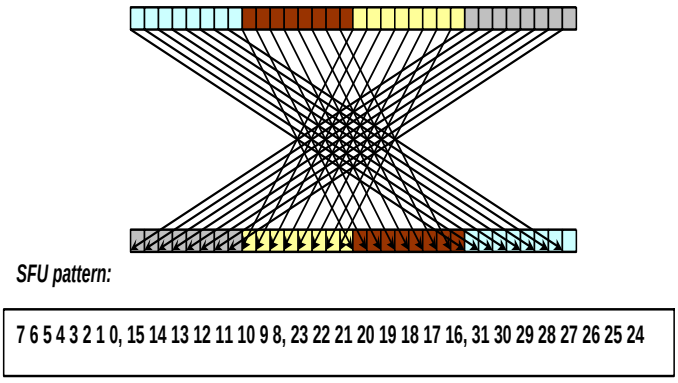

Fig. 11. SFU operation to restore pixel order for TR/BR

since the six HBs are stored in the AMU registers, vectors are frequently fetched from the AMU to the SFU. Each such fetch is represented as a separate AMU operation and requires one cycle. Together with BLIP (8 MAC and 2 bit-shift), Clip (2 MAX and 2 MIN) and SAD ((SUB×2+MAX+DIADD)×2+ADD), there are 51 AMU operations to be executed, and AMU becomes the critical resource. ME5 offers a speedup of 1.46 over ME1, requiring 107 cycles.

V. ME IMPLEMENTATION FOR PSO-BASED FRAME

In this section we show that for a PSO-based frame the pixels that are needed to construct the blocks used in BLIP are located in a much more regular pattern than for a HB-based frame. This allows to use shift units SLU and SRU instead of shuffle unit SFU in order to extract these pixels and, consequently, a more efficient ME implementation. Since the implementation of BLIP, Clip, and SAD do not depend on the storage format, our discussion will be focused on construction of the input blocks for BLIP.

We recall that in 3DRS algorithm the search region is 9×5 blocks (72×40 pixels) and each row of it is stored in 3 vectors (96 words) in the VM. The first two vectors and the first (rightmost) 8 words of the third vector contain a row's data. Four blocks TL, BL, TR, and BR needed for BLIP (see Figure 4) are contained in the square of 9×9 pixels. Therefore, in the three vectors which hold a row, there are 9 consecutive pixels that are useful for BLIP. Blocks can be fetched by using the SFU, but the efficiency of the SFU is still quite low, about 9/32. However, since the pixels are consecutive, shift operations can be used. For example, to construct TL/BL blocks we employ shift-left (SLU) unit in a following way.

A vector from the VM is fetched so that 9 consecutive useful pixels are located at the leftmost positions within the vector, as depicted in Figure 10. Then, the SLU operation is executed to obtain the useful data. One vector contains 9 pixels useful for BLIP. In order to obtain a row (8 pixels) of a TL/BL block, quad-word shifts are used, as depicted in Table II. We remark that in this code fragment the scalar and the vector receive operations (*SRCV* and *RCV*, respectively) write data to the register file of the same SLU unit which produces the data to be written. Because of this, the *RCV* and *SRCV* can be scheduled at the same cycle with the producing (i.e., shift) operations. After executing the code depicted in Table II four times, the *slu1* contains data of one half-block for BLIP. Then MAC operations are executed to perform BLIP, and at the same time, data are fetched from the VM to form new half-block in the *slu1*.

Similarly to TL/BL, the TR and BR blocks can be constructed by loading vectors, so that 9 required pixels are located at the rightmost positions, and applying subsequently shift-right operations. In this case, however, the order of pixels is permuted and the extra SFU operation (shown in Figure 11) is needed. We remark that SRU and SFU operations which form TR/BR are independent of the SLU operations which form TL/BL and, therefore, can be done in parallel with them. Hence, the work performed solely by SFU for HB-based frames, is now distributed to two functional units and significant speedup can be expected.

	VMU/UR	SFU/UR	SLU/UR	SRU/UR	NoC/ME1*
ME1	55/35%	**78/50%**	0/0%	0/0%	156/100%
ME6	28/41%	8/12%	**35/52%**	**35/52%**	68/229%
ME7	25/35%	0/0%	**35/49%**	**35/49%**	72/217%
ME8	**43/61%**	8/11%	18/26%	18/26%	70/223%
ME9	**27/50%**	8/15%	9/17%	9/17%	54/289%

TABLE III
PERFORMANCE OF ME FOR A PSO-BASED FRAME

Results

Table III summarizes the performance results of the following ME implementations for a PSO-based frame. In the following, we describe in detail different ME implementations for a PSO-based frame.

ME6: This is the ME implementation described above: SLU unit is used to form the TL/BL blocks and SRU together with SFU unit are used to form the TR/BR blocks. As expected, due to distribution of block construction task to SLU and SRU, ME6 attains an impressive 2.29 speedup with respect to the baseline HB-based implementation ME1.

ME7: This implementation is similar to ME6. However, while the data processed by SLU to form TL/BL is loaded as usual, from top to bottom, the data, which is needed to construct TR/BR by SRU operations, is loaded from bottom to top. In this way the pixels in the vectors produced by SRU are not permuted and the SFU operation needed by ME6 to rearrange the shifted data is avoided. The execution time is 72 cycles, close to that of ME6. That is because most of the SFU operations in ME6 has been overlapped with other FUs operations and they are not critical operations.

ME8: This ME implementation greatly differs from ME6 and ME7. Instead of loading 32-pixel vector and then extracting those data that belong to a BLIP input block, we suggest assembling the block by loading 4-pixel scalar values from the VM consecutively using the SLU or SRU. Since at most 4 pixels (one quad word) can be fetched from the VM at one time in the current CVP, ME8 fetches one quad word scalar at a time and shifts it into a vector using the SLU or the SRU to form one vector (half-block) for BLIP. In total, there are 9×2 SLU and 9×2 SRU operations and $9 \times 2 \times 2$ VMU quad word scalar load (*qw_scalar_send*) operations. As can be seen from Table III, ME8 is an efficient approach realized in 70 cycles. The critical resource is the VMU.

ME9: We propose to extend CVP to allow 8-word scalar operations, i.e., to introduce 8-word scalar load from VM and 8-word scalar shift operation in the SLU and the SRU. This approach allows to reduce the number of VMU/SLU/SRU operations. To construct the input blocks fro BLIP, 9×2 8-word scalar VMU loads and 9 8-word scalar shift-in operations for SLU and the SRU are executed. Note that all the other operations are the same to those in ME8. The schedule takes 54 cycles and achieves a speedup of 1.26 over ME8 and of 2.89 over ME1. We remark, however, that in order to implement ME9, the datapath width of the scalar section should be increased. Widening of the datapath increases the hardware costs and should be carefully considered before being implemented.

VI. VECTORIZATION OF (I)DCT

In this section, mapping of (I)DCT algorithm on the CVP is presented. In MPEG4 algorithm, after finishing motion estimation and motion compensation of one block, 2D *Discrete Cosine Transform (DCT)* is performed. In the decoding process, 2D inverse DCT (IDCT) is performed to decompress the data. The DCT and IDCT are very similar and, therefore, the discussion is focused only on the DCT implementation on CVP.

The 2D DCT is performed on 8×8 blocks and has separable nature, i.e., first, the 1D DCT on each of the eight columns is performed, followed by a 1D DCT on each of the eight rows:

$$y_{kl} = \frac{C(k)}{2} \sum_{i=0}^{7} \left(\left[\frac{C(l)}{2} \sum_{j=0}^{7} x_{ij} \cos(\frac{(2j+1)l\pi}{16}) \right] \cos\left(\frac{(2i+1)k\pi}{16} \right) \right)$$

Let X be an 8×8 matrix representing the input block for DCT, C be the block of DCT coefficients, Z – the block of intermediate values, and Y – the result block. Using this matrix notation, DCT can be expressed by the following two equations:

$$Z = C \cdot X, \; Y^T = Z^T \cdot C \tag{3}$$

We remark that it is acceptable for DCT to compute Y^T instead of Y because after the DCT the zig-zag scan is performed on Y, and its direction can be easily altered to operate on Y^T instead of Y. In this way, one matrix transposition can be avoided. The algorithm thus consists of matrix-matrix multiplication, transposition, and another matrix-matrix multiplication, as shown in Figure 12.

To compute the matrix product $Z = C \cdot X$, the dot product of each row of X with each column of C should be computed. Since the CVP does not contain instruction which can calculate the dot product of two vectors in parallel, matrix multiplication cannot

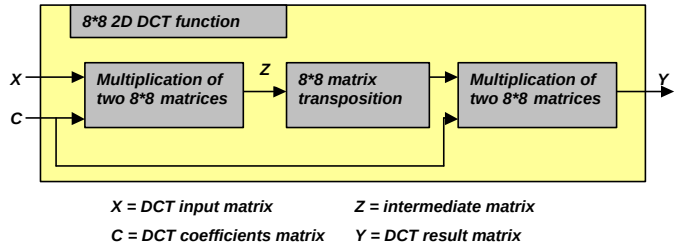

Fig. 12. Functional blocks of 2D DCT

	VMU/UR	AMU/UR	SFU/UR	NoC/DCT1*
DCT1	44/37%	43/36%	**64+32(cfg)/80%**	120/100%
DCT2	44/48%	43/47%	**32+32(cfg)/70%**	92/130%
DCT3	18/20%	43/47%	**64/70%**	92/130%
DCT4	18/28%	**43/66%**	32/49%	65/185%

TABLE IV
2D DCT PERFORMANCE OF DIFFERENT APPROACHES

be implemented straightforwardly. However, the CVP contains *multiply and accumulate* (MAC) instruction. This instruction has format MAC(*scr1*, *scr2*, *acc*), and performs the following operations: $acc_i+ = scr1_i \times scr2_i, i = 1, 2, \ldots, 32$, where *scr1* and *scr2* designate CVP registers that are 256 bits and contain 32 8-bit values, and *acc* designates a CVP accumulator which consists of 512 bits and contains 32 16-bit values. MAC cannot be utilized straightforwardly: suppose the first row of X is contained in *scr1* and the first column of C is in *scr2*. Then, the first MAC will compute products of $(x_{00}c_{00}, x_{01}c_{10}, \ldots, x_{07}c_{70})$ in parallel and store them in the elements $(acc_0, acc_1, \ldots acc_7)$ of the accumulator register. However, to obtain z_{00}, these elements have to be accumulated, which cannot be done efficiently.

Therefore, the algorithm is modified such that, for example, all the partial products needed to compute z_{00} will be contained in the first elements of the vectors, partial products needed for z_{01} in the second elements of the vectors, etc. This is achieved by permuting the input matrix data using SFU unit. For brevity, we do not describe these operations here. In the following, we evaluate the performance of the presented baseline DCT algorithm, referred to as **DCT1**, and of its optimized versions **DCT2, DCT3**, and **DCT4**. and propose several optimizations.

Results

Table IV summarizes the performance results for the following DCT implementations on the CVP.

DCT1: This is the baseline DCT implementation described earlier. We observe that SFU is the critical resource, which executes 96 operations and has utilization of 80%. There are two reasons for such a high number of SFU operations. First, in order to prepare input matrices for the MAC operations 32 different shuffle patterns should be used, resulting in 32 loads to SFU configure registers. Second, since full shuffles in current CVP require an even and an odd shuffles to be executed, $32 \cdot 2 = 64$ SFU shuffles are needed to perform 32 full shuffles with different patterns.

DCT2: This algorithm is a simple modification of DCT1 under assumption that a full vector shuffle can be executed as a single SFU operation. This allows to reduce the pressure on SFU and achieve a speedup of 30%. However, the SFU remains the critical resource. We remark, furthermore, that DCT2 cannot be implemented on the current CVP.

DCT3: As was stated in the Section IV (see discussion of the **ME2** algorithm), if the SFU can receive data from itself directly (i.e., if a datapath from the SFU output (*sfu_state*) to the SFU input (*sfu_input*) is added as presented in Figure 9), the output of the SFU can be immediately sent to the SFU input register in the same cycle. If this changes to SFU are made, less shuffle patterns and, consequently, less pattern load operations are needed. The corresponding modification of DCT algorithm, referred as DCT3 requires 92 cycles and achieves a 30% speedup with respect to DCT1.

DCT4: This algorithm combines the optimizations used in DCT2 and DCT3. It achieves a 85% speedup over the DCT1. We remark also that the critical resource for DCT4 is AMU, not SFU.

VII. Conclusions and Future Work

In this paper we have studied feasibility of vectorization of the MPEG4 kernels on the CVP. We have shown that motion estimation and (I)DCT, which are the most computationally intensive parts of the MPEG4 algorithm, can be vectorized and hence, the data-level parallelism can be exploited. Furthermore, we have shown that, usually, several vector operations can be executed in parallel, exploiting the instruction-level parallelism. Several approaches exploiting both data and instruction-level parallelism for ME and DCT have been presented.

For ME, we have studied two different storage organizations of the video frame, which are Half-Block-Based (HB-Based) storage and Pixel-Scan-Order-Based (PSO-Based) storage.

The performance of ME implementation on the current CVP for each of these storage schemes is pre-

sented in Table V. In the second column of this table the number of cycles (NoC) required for ME is depicted showing that . the ME implementation using PSO-Based scheme (ME6) provides higher performance with a speedup of 2.24 over the ME1, an implementation for an HB-Based frame. The third column in Table V depicts the performance in terms of *Millions Instructions per Second (MIPS)*, under assumption that the video sequence has the CIF format and the frame rate of 30 frames per second. A DCT

	NoC	Performance(MIPS)
HB-Based(ME1)	156	45.4
PSO-Based(ME6)	68	20.3

TABLE V
PERFORMANCE FOR MOTION ESTIMATION IN TWO DIFFERENT STORAGE ORGANIZATIONS

implementation on the current CVP, called DCT1, requires 120 cycles, or $5,7$ MIPS.

Several modifications to the current CVP architecture have been proposed in order to improve the performance of ME and DCT. The most promising suggestions are ME9 and DCT4. The ME9 algorithm requires CVP to be extended to allow operations on 8-word elements and achieves a 25% speedup over ME6 and a speedup of 2.79 over ME1. To implement DCT4 a full shuffle operation is needed, as well as a datapath from the SFU output to the SFU input. It achieves a speedup of 1.9 over DCT1.

Although we have accomplished vectorization of motion estimation and DCT and obtained remarkable speedups, there are several possibilities for future work. To improve performance of motion estimation, we can employ non-unit stride memory access technique[3] to fetch video blocks from vector memory in a clever way. This requires a modification on vector memory addressing. The data from one half-block can be fetched into one vector at one time. In other words, no data will be discarded after data fetching and the data fetching efficiency will increase significantly.

There exist several fast DCT algorithms[4], [5], [6] which reduce the number of additions and multiplications by exploiting the special structure of the coefficient matrix. Mapping these algorithms on the CVP is likely to provide further performance improvements for DCT.

REFERENCES

[1] C.H (Kees) van Berkel, Patrick P.E. Meuwissen, Nur Engin and S. Balakrishnan. CVP: A Programmable Co Vector Processor for 3G Mobile Baseband Processing. *World Wireless Congress*, 2003.

[2] Kees van Berkel, Patrick Meuwissen, Sander Weijs, Rob Wubben and Nur Engin. Obelix: a Co Vector Processor for 3rd Generation Mobile Communication: An Architecture Study. Technical Report 2001/031, Philips Research Laboratories, Eindhoven, The Netherlands, August 2002.

[3] Hennessy, John L. and Patterson, David A. *Computer Architecture – A Quantitative Approach.* Morgan Kaufmann Publishers, third edition, 2002.

[4] Byeong Gi Lee. A New Algorithm to Compute the Discrete Cosine Transform. *IEEE Trans. On Acoustics, Speech, and Signal Processing*, ASSP-32:1243–1245, 1984.

[5] Nam Ik Cho and Sang Uk Lee. Fast Algorithm and Implementation of 2-D Discrete Cosine Transform. *IEEE Trans. Circiuts Syst.*, CAS-40:259–266, April 1991.

[6] Yuh-Ming Huang and Ja-Ling Wu. A Refined Fast 2-D Discrete Cosine Transform Algorithm. *IEEE Trans. on Signal Processing*, 47(3):904–907, March 1999.